Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/166155

Please be advised that this information was generated on 2021-09-02 and may be subject to change.

A new hope on ARM Cortex-M

Erdem Alkim[1], Philipp Jakubeit[2], and Peter Schwabe[2] ⋆

[1] Department of Mathematics, Ege University, Turkey
erdemalkim@gmail.com
[2] Digital Security Group, Radboud University, The Netherlands
phil.jakubeit@gmail.com, peter@cryptojedi.org

Abstract. Recently, Alkim, Ducas, Pöppelmann, and Schwabe proposed a Ring-LWE-based key exchange protocol called "NewHope" [2] and illustrated that this protocol is very efficient on large Intel processors. Their paper also claims that the parameter choice enables efficient implementation on small embedded processors. In this paper we show that these claims are actually correct and present NewHope software for the ARM Cortex-M family of 32-bit microcontrollers. More specifically, our software targets the low-end Cortex-M0 and the high-end Cortex-M4 processor from this family. Our software starts from the C reference implementation by the designers of NewHope and then carefully optimizes subroutines in assembly. In particular, compared to best results known so far, our NTT implementation achieves a speedup of almost a factor of 2 on the Cortex-M4. Our Cortex-M0 NTT software slightly outperforms previously best results on the Cortex-M4, a much more powerful processor. In total, the server side of the key exchange executes in only 1 467 101 cycles on the M0 and only 860388 cycles on the M4; the client side executes in 1 738 922 cycles on the M0 and 984 761 cycles on the M4.
Keywords. Post-quantum key exchange, Ring-LWE, embedded microcontroller, NTT.

1 Introduction

Almost all asymmetric cryptography in use today relies on the hardness of factoring large integers or computing (elliptic-curve) discrete logarithms. It is known that cryptography based on these problems will be broken in polynomial time by Shor's algorithm [25] once a large quantum computer is built. It is, however, unknown when this will be achieved. Researchers

⋆ This work has been supported by TÜBITAK under 2214-A Doctoral Research Program Grant and 2211-C PhD Scholarship, by Ege University under project 2014-FEN-065, by the European Commission through the Horizon 2020 program under project number ICT-645622 (PQCRYPTO), and by Netherlands Organization for Scientific Research (NWO) through Veni 2013 project 13114. Part of the work was done while Erdem Alkim was visiting Radboud University. Permanent ID of this document: c7a82d41d39c535fd09ca1b032ebca1b. Date: 2016-07-19

from IBM estimate the arrival of such quantum computers within the next 2 decades [27]. This does not only imply that we need to switch to so-called *post-quantum cryptography* in 15 or 20 years. For content that we want protected over a period of 15 years or longer it is a necessary to switch already today. This has been recognized, for example, by the NSA [1], by NIST [19], or by the Tor project [16].

In the majority of contexts the most critical asymmetric primitive to upgrade to post-quantum security is ephemeral key exchange. In 2015, Bos, Costello, Naehrig, and Stebila proposed a post-quantum key exchange based on the Ring-learning-with-errors (RLWE) problem for TLS [7]. Later in 2015 (with updates in 2016), Alkim, Ducas, Pöppelmann, and Schwabe significantly improved on this proposal (in terms of speed, message size, and security) with a protocol that they call NewHope. This protocol is now used in a post-quantum-crypto experiment by Google [8] and is considered as one option to upgrade Tor's handshake to post-quantum cryptography. See [16, Slide 16] and [14]. In Section 2.3 of the 2015-12-07 version of [2], the authors of NewHope state that

> "it [...] can be implemented in constant time using only integer arithmetic - which is important on constrained devices without a floating point unit."

In this paper we present such an implementation of NewHope on "constrained devices"; specifically on the ARM Cortex-M0 and the ARM Cortex-M4 microcontrollers. Our software starts from the C reference implementation by Alkim, Ducas, Pöppelmann, and Schwabe and then carefully optimizes all performance-critical routines in ARM assembly.

Contributions. Our software is to our knowledge the first to achieve 128 bits of post-quantum security (with a comfortable margin) for key exchange on an embedded microcontroller. In terms of speed, the software is not only competitive, but actually considerably faster than today's elliptic-curve-based solutions. For example, our software outperforms the Curve25519 [4] implementation for the Cortex-M0 presented in [11] by more than a factor of two.

This speed is possible in part because of the design of NewHope, and in part through a careful optimization of the software on the assembly level. In particular for the number-theoretic transform (NTT) we show significant speedups that will also be useful in implementations of other lattice-based schemes. Specifically, our dimension-1024 NTT takes 87 223 cycles on the Cortex-M4. The previous speed record on this architecture was 71 090 cycles for a dimension-512 NTT from [9]. An NTT is essentially

a sequence of "butterfly" operations where the number of butterflies is $n \cdot \log(n)$ for a dimension-n NTT. One would thus expect the number from [9] to scale up to $10/9 \cdot 2 \cdot 71\,090 = 157\,977$ cycles, almost a factor of two slower than our result. On the much more restricted Cortex-M0 our NTT needs only 148 517 cycles and thus still outperforms the (scaled) result from [9]. Other components that we optimized on the assembly level include the error reconciliation [2, Section 5] and the ChaCha20 stream cipher [5] that is used for efficient generation of uniform noise.

Availability of the software. We place all of the software described in this paper into the public domain to maximize reusability of our results. It is available at `https://github.com/newhopearm/newhopearm.git` and `https://github.com/erdemalkim/newhopearm`.

Organization of this paper. Section 2 describes the NewHope post-quantum key exchange scheme. Section 3 gives a brief overview of the Cortex-M processor family and zooms into the specifications of and differences between the Cortex-M0 and the Cortex-M4. Section 4 provides detailed information of design decisions and constraints for both target devices. Finally, Section 5 presents and discusses our results and compares them to previous work.

2 The NewHope RLWE-based key exchange

The NewHope key exchange protocol is an instantiation of Peikert's RLWE-based passively secure KEM presented in [22]. This section recalls the specification of the key exchange and in particular explains the computations involved in the subroutines that our software optimizes on the ARM Cortex-M0 and the Cortex-M4. For a detailed motivation of the design choices in NewHope and a security analysis see [2].

The high-level overview of NewHope, as also listed in [2, Protocol 4], is given in Protocol 1. In this overview, all elements printed in boldface, except for $\mathbf{r}$, are elements of the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$, where $q = 12289$ and $n = 1024$. The element $\mathbf{r}$ is in $\{0, 1, 2, 3\}^n$. The operation $\circ$ denotes pointwise multiplication. All other operations are explained in more detail in the following paragraphs.

Parse(SHAKE-128). NewHope generates a new (public) parameter $\mathbf{a}$ for each key exchange. This eliminates concerns about backdoors in this parameter and all-for-the-price-of-one attacks (see [2, Section 3]). Server-side applications are free to cache this parameter for several key exchanges to improve performance, but our software, like the reference implementation,

Parameters: $q = 12289 < 2^{14},\ n = 1024$ Error distribution: ψ_{16}^n		
Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0,\ldots,255\}^{32}$		
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$		
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$	$\xrightarrow[1824\ \text{Bytes}]{m_a = \mathsf{encodeA}(seed, \hat{\mathbf{b}})}$	$(\hat{\mathbf{b}}, seed) \leftarrow \mathsf{decodeA}(m_a)$
		$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$
		$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
		$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
		$\mathbf{v} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow \mathsf{decodeB}(m_b)$	$\xleftarrow[2048\ \text{Bytes}]{m_b = \mathsf{encodeB}(\hat{\mathbf{u}}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \mathsf{HelpRec}(\mathbf{v})$
$\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		$\nu \leftarrow \mathsf{Rec}(\mathbf{v}, \mathbf{r})$
$\nu \leftarrow \mathsf{Rec}(\mathbf{v}', \mathbf{r})$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$		

Protocol 1: The NewHope protocol including NTT and NTT^{-1} computations and sizes of exchanged messages; $\circ$ denotes pointwise multiplication; $x \xleftarrow{\$} \chi$ denotes the sampling of $x \in \mathcal{R}$ according to χ if χ is a probability distribution over $\mathcal{R}$; $a \xleftarrow{\$} \mathcal{R}_q$ denotes the uniform choice of coefficients from $\mathbb{Z}_q$; $y \xleftarrow{\$} \mathcal{A}$ denotes that the output of $\mathcal{A}$ is assigned to y where $\mathcal{A}$ is a probabilistic algorithm running with randomly chosen coins.

does not include caching. The parameter $\mathbf{a}$ is generated from a random 32-byte seed by extending this seed through the SHAKE-128 extendable-output function (XOF) from the FIPS-202 standard [21]. The output of SHAKE-128 is considered as an array of 16-bit little-endian unsigned integers. Each of these integers is used as a coefficient of $\mathbf{a}$ if it is smaller than $5q = 61445$. Note that the amount of SHAKE-128 output required to "fill" all coefficients of $\mathbf{a}$ may differ for different seeds (because a different amount of 16-bit integers may be discarded). This is not a problem, because a XOF is designed to produce outputs of variable length. It is also not a problem from a side-channel perspective, because $\mathbf{a}$ is public.

Sampling noise polynomials from ψ_{16}. The distribution ψ_k is a centered binomial, which is used as LWE secret and error. NewHope uses the parameter $k = 16$. The distribution ψ_{16} has a mean of 0 and a variance of 8, which leads to the standard deviation of $\sigma = \sqrt{8}$. Generating a noise polynomial requires secure random-number generation. For this

purpose we use the ChaCha20 stream cipher [5] to expand a 32-byte seed (or, optionally on the Cortex-M4, the built-in hardware RNG).

NTT and NTT^{-1}. The core computational effort of NewHope lies in the number-theoretic transforms (NTTs), which are to a large extent inherently embedded into the protocol, because the exchanged messages contain polynomials in the NTT domain. The NTT transform has three sub-routines: pointwise multiplication, bit reversal of the coefficients of the polynomials, and the NTT calculation itself. All input polynomials have randomly chosen coefficients, therefore, we can assume that the coefficients are already in bit-reversed order. This leads to the situation, where our forward transform consists only of the NTT and multiplication by square roots of twiddle factors. The NTT^{-1} consists of the transform, the multiplication by the square roots of the twiddle factors and a bit-reversal.

Encoding of messages. The key-exchange requires two message exchanges by the corresponding two parties, as can be seen in Protocol 1. The main part of each message is a 1024-coefficient polynomial with 14-bit coefficients. Those polynomials are encoded into a compressed little-endian array, which takes a total of 1792 bytes. The message m_a contains an additional 32-byte seed and thus reaches a total size of 1824 bytes; m_b contains additional 256 bytes of reconciliation information and thus reaches a total size of 2048 bytes.

Rec and HelpRec. The Error reconciliation of NewHope is based on finding the closest vector in a 4-dimensional lattice with basis

$$B_4 = \begin{pmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0.5 \end{pmatrix}.$$

The HelpRec first splits the 1024 coefficients of the input polynomial $\mathbf{v}$ into 256 4-dimensional vectors $\mathbf{x}_i = (\mathbf{v}_i, \mathbf{v}_{i+256}, \mathbf{v}_{i+512}, \mathbf{v}_{i+768})^t$, for $i = 0, \ldots, 255$. It then computes reconciliation information $\mathbf{r}_i$ from those $\mathbf{x}_i$ as

$$\mathbf{r}_i = \mathsf{HelpRec}(\mathbf{x}_i, b) = \mathsf{CVP}_{D_4}\left(\frac{2^r}{q}(\mathbf{x}_i + b\mathbf{g})\right) \bmod 2^r,$$

where b is a random bit and $\mathbf{g} = (0.5, 0.5, 0.5, 0.5)^t$. Algorithm 1 describes the computation of the closest vector denoted as CVP_{D_4}. Note that the output of HelpRec as stated above is a 4-dimensional vector with entries in $\{0, 1, 2, 3\}$ (i.e., 2-bit entries). Application to the whole polynomial $\mathbf{v}$

means applying it 256 times (for all $\mathbf{x}_i$). This produces a total of 2048 bits of reconciliation information.

Algorithm 1 $\mathsf{CVP}_{D_4}(\mathbf{x} \in \mathbb{R}^4)$

Ensure: An integer vector $\mathbf{z}$ such that $\mathbf{Bz}$ is a closest vector to $\mathbf{x}$
1: **if** $(\|\mathbf{x} - \lfloor \mathbf{x} \rceil\|_1) < 1$ **then**
2: **return** $(\lfloor x_0 \rceil, \lfloor x_1 \rceil, \lfloor x_2 \rceil, 0)^t + \lfloor x_3 \rceil \cdot (-1, -1, -1, 2)^t$
3: **else**
4: **return** $(\lfloor x_0 \rfloor, \lfloor x_1 \rfloor, \lfloor x_2 \rfloor, 1)^t + \lfloor x_3 \rfloor \cdot (-1, -1, -1, 2)^t$
5: **end if**

The Rec function also works on 4-dimensional vectors and is defined as $\mathsf{Rec}(\mathbf{x}, \mathbf{r}) = \mathsf{LDDecode}(\frac{1}{q}\mathbf{x} - \frac{1}{2^r}\mathbf{Br})$, where $\mathsf{LDDecode}$ is given in Algorithm 2 (see [2, Algorithm 2]).

Algorithm 2 $\mathsf{LDDecode}(\mathbf{x} \in \mathbb{R}^4/\mathbb{Z}^4)$

Ensure: A bit k such that $k\mathbf{g}$ is a closest vector to $\mathbf{x} + \mathbb{Z}^4$: $\mathbf{x} - k\mathbf{g} \in \mathcal{V} + \mathbb{Z}^4$
1: $\mathbf{v} = \mathbf{x} - \lfloor \mathbf{x} \rceil$
2: **return** 0 if $\|\mathbf{v}\|_1 \leq 1$ and 1 otherwise

The divisions by q and the presence of values like $1/2$ might suggest that the computation of the $\mathsf{HelpRec}$ and Rec requires floating-point arithmetic. However, one can simply multiply all values by $2q$ to obtain integers; this is what the authors of NewHope refer to as efficiently implementable in fixed-point arithmetic.

Operation costs of NewHope. Table 1 summarizes the operations involved on either side of the NewHope key exchange.

Table 1: Operation counts on the client and the server side of NewHope.

Operation	**Server**	**Client**
Generating the public parameter $\mathbf{a}$;	1	1
Sampling noise polynomials;	2	3
Computing the NTT;	2	2
Computing the NTT^{-1} with bit reversal;	1	1
Computing the pointwise multiplication;	2	2
Computing the vector $\mathbf{r}$ for error reconciliation;	0	1
Computing the error reconciliation Rec;	1	1
Hashing the 32-byte value ν with SHA3-256 to obtain the final key μ.	1	1

3 The Cortex-M family of microcontrollers

The ARM Cortex-M processors are advertised as "the most popular choice for embedded applications, having been licensed to over 175 ARM partners" [15]. Their wide deployment in embedded applications makes them an attractive target for optimized cryptography. ARM offers a wide range with their Cortex-M family. At the low end of pricing, power consumption, and also computational capabilities is the Cortex-M0. At the high end are the Cortex-M4 and Cortex-M7. Like other embedded processors, ARM Cortex-M chips are used in the Internet of Things, consumer products, medical instrumentation, connectivity, or industry-control systems.

All Cortex-M processors have in common that data is processed in 32-bit words. Relevant differences for the software described in this paper are the instruction set, the size of RAM and ROM, and the availability of a random source. The Cortex-M0 is based on the ARMv6-M architecture. This architecture combines the 16-bit Thumb instruction set with a few 32-bit instructions. The Cortex-M4 is based on the ARMv7-M architecture. This architecture makes use of the 32-bit Thumb-2 instruction set. Both processors have 16 general-purpose registers, out of which one is used as stack pointer (`r13`), one is used as link register (`r14`), and one for the program counter (`r15`). However, only 32-bit instructions can make use of the 8 high general-purpose registers, which limits the Cortex-M0 to essentially eight general purpose registers (except for register-to-register copies, which can also reach the high registers). Another difference con-

cerns the size of immediate values that instructions can handle: The M0 instruction set supports only 8-bit immediate values; the M4 instruction set supports immediate values of up to 16 bits.

Both processors have a comparable timing with respect to cycle count of atomic instructions. For example, the branch instruction needs 3 cycles if the branch is taken and 1 cycle otherwise on both architectures. Both architectures provide instructions to load or store multiple registers in $1 + n$ cycles, where n is the number of registers. In the case of load and store instructions, however, architectural differences occur. On the Cortex-M4, store instructions take only one cycle, because address generation is performed in the initial cycle and the actual storing of data is performed while the next instruction is executed. Load instruction can be pipelined together with other independent load and store instructions. The Cortex-M0 does not provide pipeline functionality for load and store instructions; those instructions thus take 2 cycles.

The Cortex-M0 does not have a hardware random-number generator (RNG), whereas the Cortex-M4 on our STM32F4xx-series development board offers a 32-bit hardware RNG. This RNG unit passes all statistical tests for secure random number generation provided by the NIST [26]. For the M4 we present two versions of our noise generation: one using ChaCha20 and one using this hardware RNG (which has also been used for noise generation in [9]).

4 Implementation details

This section first provides a detailed explanation of general optimization techniques. We then provide two architecture-specific subsections in which we elaborate on processor-specific optimization techniques. For the SHAKE-128 function and the SHA3-256 function we use the optimized implementation by the Daemen, Peeters, Van Assche and Van Keer [6].

The main focus of our optimization lies on the NTT and the NTT^{-1}. In our description we treat the NTT and the NTT^{-1} together, because they only differ in the fact that the NTT^{-1} requires a bit reversal and in the constants being used: powers of ω for the NTT and powers of ω^{-1} for the NTT^{-1}. The choices for these parameters made by the designers of NewHope are $\omega = 49$ and $\omega^{-1} = 49^{-1} \mod q = 1254$. This implies that $\gamma = 7$ is the square root of ω, the n-th root of unity. The existence of ω and γ is guaranteed by the parameter choice of $n = 1024$ and $q = 12289$, which is the smallest prime for which $q \equiv 1 \mod 2n$. This together with n being a power of 2 allows an efficient implementation of the NTT for elements

of $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. As an obvious optimization we make use of precomputed powers of ω and γ, and removed multiplications by $\omega_0 = 1$ from last level of the $\mathbb{N}$TT. These well known optimization techniques for speeding up the NTT computation save us 1525 multiplications.

For precomputing the constants, there are essentially three different strategies to trade-off time and memory. One approach is to precompute none of the powers of ω and γ the other extreme is obviously to precompute all of the powers of ω and γ; a middle ground is to precompute a subset of them. Not precomputing any powers implies that only one coefficient needs to be stored and the rest is generated 'on the fly', which costs one additional multiplication per power. The cost intensive aspect, however, is that the product needs to be reduced afterwards, which rules out this option for us as we chose to focus on efficiency. Precomputing all powers was the logical approach to begin with due to consistency with the reference implementation provided by [2]. This requires to store 3072 14-bit coefficients: the 512 powers of ω, the 512 bit reversed powers of ω, the 1024 powers of γ, and the 1024 inverted powers of γ. These constants, however, have a partial overlap, which points into the direction of the third approach, namely to balance the memory usage and the computational costs. We found in our experiments that the most balanced approach is to store the 512 powers of ω and use them to compute the powers of γ. The first 512 elements of the powers of γ are identical to the powers of ω, because the powers of γ are bit reversed. The second 512 elements can be computed by a simple multiplication with $\gamma = 7$. Since 7 needs only 3 bits and both the precomputed powers of ω and the coefficients are 14-bits in size no reduction is required, because we operate on a 32-bit architecture and after a multiplication the maximum bit size is $3 + 14 + 14 = 31$-bits. With this approach we were able to reduce the size of precomputed tables needed by a factor of $\frac{1}{3}$ for a price of ≈ 750 cycles. It is the most efficient setup for the NTT transform with regards to both memory and computational costs, as it only requires to keep 512 14-bit coefficients at a low cycle count overhead.

The approach for NTT^{-1} it is not as straight forward, because the powers of ω^{-1} are not as easily related to the powers of γ ($\gamma^{2n} \equiv 1 \mod q$). The only balancing technique we could apply would be to use same powers of ω used for the NTT. This would imply that the resulting polynomial would be in reversed order. We would then need to reorder the polynomial to the natural form. This could be integrated into the required multiplication with the precomputed powers of γ. We implemented it during our experiments and decided against it in the final implementation as it saves

only $\frac{1}{6}$ of the table sizes, (namely 512 inverted powers of ω) but introduces an overhead of $> 3\,000$ cycles. Therefore, we decided to keep the reversed powers of ω. In our speed-optimized implementation we decided against this tradeoff, but it might well be worth considering if memory constraints are an issue.

Listing 1 Reduction routines used in the butterfly operation.

(a) Montgomery reduction ($R = 2^{18}$).

```
montgomery_reduce,rm:
 MUL rt, rm, #12287
 AND rt, rt, #262143
 MUL rt, rt, #12289
 ADD rm, rm, rt
 SHR rm, rm, #18
```

(b) Short Barrett reduction.

```
barrett_reduce, rb:
 MUL rt, rb, #5
 SHR rt, rt, #16
 MUL rt, rt, #12289
 SUB rb, rb, rt
```

Listing 2 Gentlemen-Sande butterfly operation - all variables are `uint16_t`.

```
 LDR ($a_{j}$),r0
 LDR ($a_{j + d}$),r1
 MOV rt,rt,r0
 ADD r0,r0,r1
 ADD rt,rt,#36867
 SUB rt,rt,r1
 LDR ($omega_t$),r1
 MUL rt,rt,r1
 barret_reduce,r0
 montgomery_reduce,rt
 STR ($a_j$),r0
 STR ($a_{j + d}$),rt
```

The NTT for $n = 1024$ consist of 10 levels, each performing 512 Gentlemen-Sande butterfly operations [12]. Each butterfly operation consists of three loads, one addition, one subtraction, one multiplication by a constant and two stores. One more addition needs to be performed to keep all coefficients in unsigned format.

Thus, except for the modular reductions, a butterfly operation requires at least 2 registers for coefficients, one temporary register, and one 16-bit immediate value. Self-evidently we carry over the optimization techniques applied to the computation of the NTT already in place in the reference implementation. These consist of speeding up the modular-arithmetic. The first optimization is to use Montgomery arithmetic [17].

This demands that all constants are stored in the Montgomery representation with $R = 2^{18}$. Our assembly version of the Montgomery reduction is given in Listing 1a. It shows that Montgomery reduction requires two 14-bit, one 18-bit, and one 5-bit immediate value, and also one temporary register. The second optimization is to use short Barrett reductions [3] for modular reductions after addition. Our assembly version of this routine is given in Listing 1b; it shows how we reduce a 16-bit unsigned integer to an integer congruent modulo q of at most 14-bits. It requires one 14-bit, one 5-bit and one 3-bit immediate values, and one additional register. The ARM instruction set does not allow immediate values as parameter in the multiply instruction on both microcontrollers. Therefore, immediate values used in multiplications must be loaded to a register first. With these conditions, each butterfly operation requires at least 4 registers. The third optimization is called 'lazy reduction'. It describes that the short Barrett reduction is only applied every second level [2]. This works, since per level at most one carry bit occurs; the short Barrett can handle up to 16-bits and the starting value is at most 14-bits in size. However, because we are computing two additions before the reduction, we need to add $3q$ (36867) before the subtraction to keep all coefficients in the unsigned format.

A note on the Longa-Naehrig approach. As a follow-up work to [2], Longa and Naehrig presented speedups to NewHope and in particular the NTT in [13]. They claim a speedup of the NTT by a factor 1.9 in the C implementation and by a factor of 1.25 in the AVX2-optimized implementation. The central idea of that paper is a specialized modular reduction routine for primes of the shape $k \cdot 2^m + \ell$ for small values of k and ℓ; in the case of NewHope those values are $k = 3$ and $\ell = 1$. This reduction routine is combined with extensive use of lazy reduction. The factor of 1.9 in the C implementation is largely explained by the fact that the software makes heavy use of 64-bit integers, which the software described in [2] explicitly avoids. Obviously, making use of 64-bit integers makes sense on AMD64 processors, but is much less efficient on the 32-bit microcontrollers targeted in this paper. The AVX2 implementation described in [13] has in the meantime been outperformed by the latest version of the AVX2 software by the NewHope authors, which uses double-precision floating-point arithmetic.

We experimented with the approach described by Longa and Naehrig on the M0 and M4 and were not able to gain any speedups. This is partly explained by the lack of 64-bit registers (and a 32×32-bit multiplier on the M0). Another reason was that we observed a slight increase in register usage, which significantly increased the required number of loads and

stores, in particular on the M0. Furthermore, the lazy-reduction approach leaves intermediate values of > 16 bits, which need to be stored to RAM before processing the next level. Using 32-bit integers for those intermediate values increases the memory usage of the NTT by 2 KB, which is prohibitive on the M0.

4.1 Cortex-M0 specific optimization

The first optimization necessary for the Cortex-M0 is to fit NewHope onto the processor. The portable reference implementation provided by the authors of NewHope and described in [2] exceeds the Cortex-M0's 8 KB of RAM. The C reference implementation of NewHope closely follows the description in Protocol 1, and makes use of 4 polynomials during key generation and 8 polynomials for the computations on the client side. Each of these polynomials is represented by its 1024 unsigned 16-bit coefficients, and thus consumes 2 KB of RAM. Even with only minimal overhead for different variables or microcontroller internal RAM usage, only up to 3 polynomials fit simultaneously into the RAM of the Cortex-M0. By restructuring the code and adapting the data types used we could fit both, the server side and the client side onto the Cortex-M0. We solved a similar issue during noise extension. On the Cortex-M0 it is impossible to have a buffer larger than 1024-byte. We therefore perform four ChaCha20 calls. This required another bit of entropy. We simply used the loop counter used for the four consecutive calls as input byte for the second element of the initialization vector for the ChaCha20 function.

After fitting the key exchange protocol into the boundaries provided by the Cortex-M0, we could start to look into optimization for speed. A general aspect regarding optimization on Cortex-M processors is that data is processed in words of 32-bits. This allows us to cut the amount of stores and loads in half for the coefficients and constants represented as unsigned 16-bit values. For the shared key and seeds, unsigned 8-bit values, the amount of load and stores is decreased by four. For logical operations on the values loaded this way, no overhead is generated. Arithmetic operations, however, produce overhead, because the 32-bit values need to be split before computation and the 16-bit values need to be merged afterwards. This costs 2 additional cycles for every load and 2 more cycles before every store.

As can be seen in the operation counts summarized in Table 1 at the end of Section 2, the NTT and the NTT^{-1} are the most frequently called operations. Since it is also the most expensive function with regards to cycle counts, it was the natural choice to begin with.

NTT and NTT^{-1}. We began our optimization of the NTT (and NTT^{-1}), by unrolling the 10 levels and standardizing the inner loops, such that every level loops 256 times and performs two Gentlemen-Sande butterfly operations per loop iteration. Performing two Gentlemen-Sande butterfly operations per iteration is beneficial, because it allows us to make the best use of the 32-bit word size of the Cortex-M family. Listing 2 shows the code for one Gentlemen-Sande butterfly operation. For the lazy reduction on every second level the Barrett reduction is omitted. Since each coefficient is a 16-bit value, we are able to load two of them per load operation. We continued our optimization by merging levels 0 and 1. Level 0 takes every element and performs the butterfly operations; level 1 takes every second element and performs the butterfly operations. If we combine both levels for efficiency we need to load two 32-bit words, thus four 16-bit coefficients. For each 2 loads we can now perform 4 combined Gentlemen-Sande butterfly operations. We perform the two butterfly operations of level 0 (without the Barrett reduction followed by the two butterfly operations of level 1 (with the Barrett reduction). One loop iteration thus handles both levels.

These four merged butterfly operations take a total of 134 cycles. Unfortunately this does not work for the other consecutive levels on the Cortex-M0. With its limited instruction set and the resulting 8 general purpose registers, the overhead gets out of proportion when merging higher levels. Therefore we get a cycle count of 96 for every even and a cycle count of 86 for every odd level. The last optimization we performed was to minimize register reordering. We went through our NTT code and optimized it such that constants and loop-counter are placed in high registers where possible to allow to make use of the Cortex-M0's full potential.

Before each call to the NTT a multiplication with the γ coefficients and after each call to the NTT^{-1} a multiplication with the precomputed γ^{-1} coefficients must be performed. We implemented the multiplication on the coefficients in assembly to benefit from the Cortex-M0's 32-bit word size. Additionally to the architectural benefit we make use of the fact that the multiplication of the coefficients with the precomputed coefficients is a simple operation and does not need too many registers. Therefore we are able to load 4 coefficients at once and also store them. With this we decreased the amount of loads and stores needed by another factor of two. We could reduce the cycle count for the multiplication of coefficients by 55.04% compared to the reference implementation.

We also decided to rewrite the pointwise multiplication of polynomials such that it makes optimal usage of the target architecture. We achieve a

56.08% decreased cycle count, compared to the reference implementation, for the pointwise multiplication by making use of the word size. We load and store two consecutive coefficients of the polynomial and apply the calculations needed on each half word. By doing so, we only call half of the iterations of the main loop.

Before the NTT^{-1} is called a bit reversal needs to be performed. We did not provide an assembly optimized version for this function. The problem is that consecutive coefficients do not necessarily get changed, which implies that we cannot benefit from the word size. We just adapted the bit reversal to not loop over the last elements which are unaffected by it.

Sampling noise polynomials. The noise seeds which form the base of the noise polynomials are not generated on the Cortex-M0. The development board we used during the implementation does not provide an RNG. Since there is no default option for random number generation on the Cortex-M0 we made the choice to allow a context-specific implementation. The randomly generated seed is crucial for the security of the key exchange, therefore, we provide an easy to replace C function in our code. The random seed gets subsequently extended by the ChaCha20 stream cipher. We based our architecture specific implementation on a ChaCha20 implementation specifically designed for the Cortex-M0 by Neikes and Samwel [18]. The core functionality of this stream cipher is optimized in assembly. Additions we made were merely in the initialization phase. Again we benefit from the 32-bit word length of the architecture, which allowed us to represent the internal variables efficiently. The reference implementation makes use of two helper functions to store and load values in little-endian, however, this aspect can be solved simply by the little-endian architecture. Therefore, we could omit the helper functions, which gives us a 10.82% decreased cycle count compared to the reference implementation.

Error reconciliation and help-vector generation. We continued our optimization with the Rec function by implementing it in assembly. This yields the general benefits of the 32-bit word size. By additionally unrolling and restructuring the loop we make even better use of the architecture. We calculate 8-bits of the key and perform four consecutive calls to this function to get 32-bit of the key before storing it. We store 32-bit of the key eight times to compute all 256 bits of the key. Contrary to the reference implementation, we apply helper functions as soon as possible without storing intermediates. These changes give us a 32.10% decreased cycle count compared to the reference implementation.

In the case of the HelpRec function, we first benefit from the fast ChaCha20 implementation. We continued by rewriting the main loop in assembly. The loop iterates over the 256 random bits used as fair coin and encodes each bit into 4 coefficients of the input polynomial. We restructured the loop to load 8 times a full word (32-bit). Afterwards, we perform the loop internal calculations per bit and apply the results to the four positions of the polynomial. These optimization measures grant us a 14.43% faster implementation compared to the reference implementation.

Polynomial addition. Additionally, we wrote assembly implementations for the basic arithmetic calculations for polynomials. The addition works by taking each coefficient of the first and each coefficient of the second polynomial at the same position and adding them together before reducing the sum with a call to the Barrett reduction. We implemented the Barrett reduction specific for the context and the architecture, such that we manage to decrease the cycle count to 5. Due to the fact that this simple function does not require meticulous register usage we could load two 32-bit words at once, thus 4 coefficients. We do so for the coefficients of the first polynomial and load 2 coefficients of the second polynomial, compute the results, load the next 2 coefficients of the second polynomial, compute the second two results and store the newly computed 4 coefficients with one instruction. We manage to reduce the cycle count required for polynomial addition by 59.02% compared to the reference implementation.

4.2 Cortex-M4 specific optimization

Compared to the Cortex-M0, the Cortex-M4 is much more powerful. It has 192KB of RAM, the portable reference implementation can thus run without adaptations on this microcontroller. Additionally, the Cortex-M4 on our development board features a hardware random-number generator. This enables us to calculate the seeds on the microcontroller directly. Additionally, we are not required to make use of *LDM* and *STM* instructions to save cycles for memory operations, thanks to the architectural benefits described in 3. This enables us to use 16-bit loads and stores directly without extracting the 16-bit coefficients from 32-bit words. The most obvious implication of this is that the C implementation performs as good as assembly when there are no arithmetic and/or reordering optimizations.

NTT and NTT^{-1}. Inside one butterfly operation, 2 temporary registers are required to calculate the results. The Cortex-M4 has 14 available general-purpose registers and we need to keep the addresses of the input

polynomial and the array of precomputed twiddle factors. Therefore, we have 10 registers available during our computations. This implies that we can merge up to 3 levels to save on loads and stores. Making use of these architectural constraints we split the NTT on the Cortex-M4 in four chunks of layers. The first two chunks each perform three layers of the NTT in one loop. These loops process 8 coefficients and run 128 times. In the third chunk we took the first 512 coefficients of the input-polynomial and ran the next three layers of the NTT on them. Afterwards, we took the second 512 coefficients of the input-polynomial and ran the same layers on them. When the results are loaded into the registers we were able to ran the last layer on them, which saved us 1024 loads and stores. The precomputed twiddle factors are such that we do not need multiplication for the last layer. We incorporate the additional register that kept the addresses of the twiddle factors into the calculations performed at the last layer. This reduces the total amount of loads and stores needed for the NTT to $3.5n$ instead of $10n$ ($n = 1024$). By applying the concept of merged layers, we where able to reduce our NTT assembly code for the Cortex-M4 to 384 branches instead of 5120 needed in the C reference implementation.

The Cortex-M4 has a 'multiply and accumulate' instruction for 32-bit integers. It can be seen that both in reductions in Listing 1 multiplication is followed by addition or subtraction. Therefore, we could use this instruction in both, butterfly and pointwise multiplication. This saves more than 30000 cycles per NTT transform. To be able to use this optimization we implemented the pointwise multiplication of polynomials in assembly.

We also implemented the bit reversal operation in assembly. However, while unrolling the bit reversal operation in assembly saves 6500 cycles, the code size of the unrolled bit reversal is 7799 bytes more than the looped implementation. Due to this trade off we decided against the use of it in our work, because we only have two NTT^{-1}'s. In another scenario, however, it could be beneficial and proofs that there is still room for improvements.

Sampling noise polynomials. We implemented the sampling of noise polynomials in two different ways on the Cortex-M4. First, we implemented the sampling by calling ChaCha20 as the reference implementation does. Second, we implemented the sampling by using the built-in RNG. It generates a 32-bit random number every 40 cycles. Each coefficient of a polynomial requires $2k$ random bits, $2k+1$ additions, $2k$ shifts, $2k$ logical 'and' instructions and 1 subtraction. For every 32-bit number we generated one coefficient in 50 cycles. These calculations take more time than required by the RNG, which implies that the RNG does not

have to wait on our calculations. Since we need 32-bit of randomness for one coefficient, the RNG is called 1024 times during the process of sampling one polynomial. As can be seen, the performance of the generation of a noise polynomial is strongly dependent on the parameter 'k'. Therefore, the running time of the noise sampling can be predicted by the time required to generate $2k$ random bits with the RNG.

The Cortex-M4 memory operation can be pipelined, thus calling two 16-bit load/store instructions takes the same amount of time as calling one 32-bit load/store instruction and split it into two 16-bit integers. This allowed us to use the C implementation for the other operations of NewHope without experiencing any significant slowdown.

5 Results and comparison

In this section, we present our results and compare them with results from the literature. Cortex-M0 benchmarks are obtained on the STM32F0 Discovery board, which is equipped with a STM32F051R8T6 microcontroller. Cortex-M4 benchmarks are obtained on the STM32F4 Discovery development board, which is equipped with a STM32F407VGT6 microcontroller. Our software is compiled with arm-none-eabi-gcc version 5.2.0 and `-Ofast` as compiler flag for both, the Cortex-M0 and the Cortex-M4. Cycle counts and ROM size of our software is summarized in Table 2.

Comparison with previous results. The literature describes various implementations of lattice-based cryptography on embedded microcontrollers.

For example, in [24] the authors targeted the AVR architecture, and in [23] the authors targeted FPGAs. A direct and fair comparison among those implementations underlies many, often unsolvable constraints. The architectures vary, different schemes are implemented, and last but not least do all candidates for comparison to our result target lower security levels. To gauge the progress of *implementation techniques*, most comparisons between different schemes focus on comparing the performance of subroutines; in the context of ideal-lattice-based cryptography mainly on comparing noise sampling and the NTT, the two most costly operations.

To the best of our knowledge, there are two papers that describe optimizations of ideal-lattice-based cryptography for the ARM Cortex-M family of microcontrollers. In [9], de Clercq, Roy, Vercauteren, and Verbauwhede optimize RLWE-based encryption and in [20], Oder, Pöppelmann, and Güneysu optimize the Bliss signature scheme by Ducas, Durmus, Lepoint, and Lyubashevsky [10]. Both papers target the Cortex-M4F

Table 2: Cycle counts of NewHope building blocks on target devices.

Operation	**Cortex-M0**	**Cortex-M4**
Generation of **a**	380 855	293 975
NTT	148 517	87 223
NTT^{-1}	167 405 [a]	97 789 [a]
Sampling of a noise polynomial	208 692 [b]	172 221 [b] (54 322)[c]
HelpRec	68 170	44 348
Rec	46 945	34 524
Key generation (server)	1 168 224	964 440 [b] (681 514)[c]
Key gen + shared key (client)	1 738 922	1 418 124 [b] (984 761)[c]
Shared key (server)	298 877	178 874
ROM usage (bytes)	30 178	22 828 [b] (18 544)[c]

[a] Includes bit reversal operation
[b] Noise generation done by *ChaCha20*
[c] Noise generation done by *RNG*

microcontroller and implemented the NTT on 512-coefficient polynomials with the same modulus $q = 12289$ that we used. An additional challenge for comparison is that the NTT operations in [9] and [20] use dimension 512, whereas we use dimension 1024. As explained in the introduction, NTT computations are essentially a sequence of butterfly operations. For comparison we thus scale the numbers from [9] and [20] to dimension 1024 by the number of butterflies, i.e., by a factor of 20/9.

From Table 3 we can see that even if we use the built-in RNG of the M4, our sampling algorithm is 1.75× slower than the Knuth-Yao algorithm used in [9]. Note however, that our sampling algorithm, unlike the Knuth-Yao sampler, runs in constant time and is thus inherently protected against timing attacks. Also, the slightly decreased performance on embedded microcontrollers is a price to pay for compatibility with significantly increased timing-attack-protected sampling performance on large processors with caches. For details, see [2, Section 4].

With respect to the NTT the cycle counts we achieve on the Cortex-M4 are 45% faster than [9] and 68% faster than [20]. In the case of the

Table 3: Performance comparison of NTT implementation and error sampling

	NTT	Noise sampling[a]
Cortex-M0 **(ours)**	148 517	270
Cortex-M4 **(ours)**	87 223	174[b] (50)[c]
Cortex-M4F [9]	157 977[d]	28.5
Cortex-M4F [20]	272 486[d]	1 828

[a] Cycle counts for sampling one coefficient
[b] Noise generation done by *ChaCha20*
[c] Noise generation done by *RNG*
[d] Number scaled from dimension 512 to dimension 1024 by multiplying by 20/9

Cortex-M0, the cycle savings are 6% faster than the M4F counts from [9] and 45% faster than the M4F counts from [20]. This demonstrates that the optimization measures applied by us provide faster results on comparable hardware and enable inferior hardware to outperform the best results on ARM Cortex-M processors for calculating a NTT.

References

1. National Security Agency. NSA suite B cryptography. https://www.nsa.gov/ia/programs/suiteb_cryptography/, Updated on August 19, 2015. 2
2. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016 (to appear). https://cryptojedi.org/papers/#newhope. 1, 2, 3, 6, 9, 11, 12, 18
3. Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *LNCS*, pages 311–323. Springer, 1987. 11
4. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519. 2
5. Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. http://cr.yp.to/papers.html#chacha. 3, 5
6. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, 2012. www.http://keccak.noekeon.org/Keccak-implementation-3.2.pdf (accessed 2016 -03-01). 8
7. Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors

problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, 2015. http://eprint.iacr.org/2014/599. 2

8. Matt Braithwaite. Experimenting with post-quantum cryptography. Posting on the Google Security Blog, 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html. 2
9. Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*, pages 339–344. EDA Consortium, 2015. http://eprint.iacr.org/2014/725. 2, 3, 8, 17, 18, 19
10. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013. https://eprint.iacr.org/2013/383. 17
11. Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Design, Codes and Cryptography*, 77(2), 2015. http://cryptojedi.org/papers/#mu25519. 2
12. W. M. Gentleman and G. Sande. Fast Fourier transforms: for fun and profit. In *Fall Joint Computer Conference*, volume 29 of *AFIPS Proceedings*, pages 563–578, 1966. http://cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf. 10
13. Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Report 2016/504, 2016. https://eprint.iacr.org/2016/504/. 11
14. Isis Lovecruft and Peter Schwabe. RebelAlliance: A post-quantum secure hybrid handshake based on NewHope. Draft proposal for Tor, 2016. https://gitweb.torproject.org/user/isis/torspec.git/plain/proposals/XXX-newhope-hybrid-handshake.txt?h=draft/newhope. 2
15. ARM Ltd. Cortex-M series, 2015. www.arm.com/products/processors/cortex-m/ (accessed 2015-12-10). 7
16. Nick Mathewson. Cryptographic directions in Tor. Slides of a talk at Real-World Crypto 2016, 2016. https://people.torproject.org/~nickm/slides/nickm-rwc-presentation.pdf. 2
17. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf. 10
18. Moritz Neikes and Niels Samwel. ARM implementation of the ChaCha20 block cipher. GitLab repository, 2016. https://gitlab.science.ru.nl/mneikes/arm-chacha20. 14
19. NIST. Workshop on cybersecurity in a post-quantum world, 2015. http://www.nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm. 2
20. Tobias Oder, Thomas Poppelmann, and Tim Güneysu. Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. ACM, 2014. https://www.sha.rub.de/media/attachments/files/2014/06/bliss_arm.pdf. 17, 18, 19
21. National Institute of Standards and Technology. FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 4

22. Chris Peikert. Lattice cryptography for the Internet. In Michele Mosca, editor, *Post-Quantum Cryptography*, volume 8772 of *LNCS*, pages 197–219. Springer, 2014. https://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf. 3
23. Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013. https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf. 17
24. Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 346–365. Springer, 2015. http://eprint.iacr.org/2015/382/. 17
25. Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997. 1
26. STMicroelectronics. AN4230 application note – STM32 microcontrollers random number generation validation using NIST statistical test suite, 2013. http://www.st.com/resource/en/application_note/dm00073853.pdf. 8
27. Jim Utsler. Quantum computing might be closer than previously thought. *IBM Systems Magazine*, 2013. http://www.ibmsystemsmag.com/mainframe/trends/IBM-Research/quantum_computing/ (accessed 2016-03-03). 2